UFR IMAG M1 Informatique
UE Conception Des Systèmes d'Exploitations
Compte Rendu TP Allocation Mémoire
LEFRERE COUTAUD

Architecture de l'allocateur:

Nous avons séparé notre code en 2 fichiers : *mem.c* qui implémente les fonctions de l'allocateur et *memRoutines.c* qui implémente une série de fonctions utilisées par *mem.c*.

Structures de données :

La gestion de la mémoire ce fait grâce à 2 listes chainées : 1 liste des blocs libres et une liste des blocs alloués. L'allocateur utilise 2 structures distinctes pour gérer ces 2 listes. Ce choix à pour but de permettre des évolutions distinctes de la gestion des blocs de mémoires alloués ou non (utiliser un liste doublement chainée pour une des 2 listes par exemple).

Fonctionnalités :

Les fonctionnalités implémentées par l'allocateur sont :

-L'initialisation de la mémoire : *mem_init*

-L'allocation de mémoire (alignée sur 16 octets pour les systèmes 64b et 8octets pour les systèmes 32b) : *mem_alloc*

-Libération de la mémoire : *mem_free*

-Consultion de la taille des blocs mémoire : *mem_get_size*

-Affichage de l'ensemble de la mémoire : *mem_show*

-Choix de la politique d'allocation entre first fit ou best fit : *mem_fit*

Tests :

Nous avons ajouté à l'allocateur une fonction *sumMyMemory* qui nous as permis de vérifier en parti le maintien de la cohérence des structures de données . Nous avons donc ajouté une assertion testant l'égalité entre la taille de la mémoire initiale et la taille des listes chainées à chaque allocation et libération dans le programme de test *memtest.c*. Nous avons testé la viabilité de notre allocateur, sur des petits volumes d'allocation mémoire avec l'application *ls* sur le répertoire du TP, puis pour des volumes d'allocation plus important avec la commande '*ls -R /*' .

Implémentation :

mem_init : La fonction mem init se résume à fixer la politique d'allocation par défault et à initialiser les pointeurs de tête des 2 listes : à NULL pour la liste des blocs alloués et sur le bloc comprenant toute la mémoire pour la liste des blocs libres.

Strategy : fonction implémentant la politique d'allocation, cette fonction nous retourne l'adresse du pointeur vers la case à allouer. Ce choix d'implémentation à l'avantage de nous permettre d'extraire le bloc en temps constant (pas de second parcours de la liste) sans avoir recours à une liste doublement chainée qui occuperai plus d'espace mémoire.

mem_alloc : Demande à la fonction implémentant la stratégie d'allocation de lui indiquer un nœud de la liste des blocs libres de capacité au moins égale à la taille demandé. Si il n'y à pas de bloc de taille suffisante, on met la variable *errno* à la valeur *EAGAIN* et on retourne NULL afin de respecter au mieux la spécification de malloc. Si le bloc est de taille suffisament grande on le scinde en 2 blocs respectant la contrainte d'alignement. On extrait ensuite le bloc choisi de la listes des blocs libres et on l'insère dans la liste des blocs alloués.

mem_free : Dans un premier temps on cherche le bloc alloué correspondant à l'adresse passée en paramêtre par l'utilisateur. Si ce bloc n'existe pas (ce n'est pas l'adresse d'un bloc

précedemment alloué) on ne fait rien de plus. Si le bloc existe, on l'extrait de la liste des blocs alloué et on l'insère dans la liste des blocs libres. L'insertion maintien la liste triée par l'ordre des adresses mémoire. On teste ensuite si les blocs libres précédent et suivant sont juxtaposés avec le nouveau bloc libre. Si c'est la cas on fusionne les blocs. L'opération est faite en même temps que l'insertion afin de faire un unique parcours de la liste.

Points faibles de notre implémentation :

Nos choix techniques ont rendu le code complexe et difficilement maintenable : Le niveau d'indirection sur les pointeurs pour le choix d'un bloc à allouer ou sur la désignation des blocs à extraire favorise grandement les erreurs de programmation avec des effets de bords non désirés et difficile à repérer. Cela rend notre programme compliqué à maintenir.

Bien que nous aillons dès le départ utilisé des structures différentes pour les blocs alloués et libre pour permettre des adaptations futures divergentes, nous avons laissé ces structures de même taille. Ce choix est mauvais à 2 égards : Non seulement l'objectif initial n'est absolument pas rempli : les cas particuliers (notamment l'alignement) du à une éventuelle différence de taille des 2 structures n'est pas géré. Mais en plus nous n'avons pas pu utilisé les mêmes routines et fonctions sur les 2 types de listes chainées : Nous aurions pu utiliser une même structure de liste chainée avec les fonctions d'ajout, d'extraction, de recherche, etc commune aux deux listes.

Points positifs de notre implémentation :

Tout d'abord nous avons essayé d'encapsuler au maximum notre code afin d'avoir des fonctions de taille raisonnable et ainsi rendre le programme facilement lisible et compréhensible. Ensuite nos choix d'implémentations bien que nous aillant compliqué la tache, permette de lilité grandement les parcours redondant de la mémoire tout en ayant des structures qui prennent relativement peu de place en mémoire.